



Performance Testing and Profiling Apps with Platform Tuning

I have not failed. I've just found 10,000 ways that won't work.

— Thomas Edison

In computing, hardware acceleration involves using computer hardware to perform functions faster than would be possible with software running on the general-purpose CPU. Normally, processors are sequential and instructions are executed one-by-one. Various techniques are used to improve processing performance, and hardware acceleration, as discussed in **Chapter 11**, is one of them. The main difference between hardware and software optimization is arguably the level of abstraction. Due to the nature of hardware optimization, they potentially provide much greater speed improvements than software optimizations. Hardware accelerators are designed for computationally intensive software code.

More and more developers use FFmpeg to develop Android video applications and OpenCV to develop image processing software, which all have NDK adaptations. Multimedia applications usually have high-performance requirements, and this chapter describes some common optimization technology on x86 Android.

Start with Your First x86 Full Format Video Player

The built-in codec program on Android is very limited, so developers use the FFmpeg free open source media framework to support full-format decoding. The FFmpeg project includes libraries of audio/video codecs and a command-line program for transcoding multimedia files, and enables cross-platform audio and video streaming using either an LGPL or GPL license according to your choice of components. It provides recording, conversion, and streaming audio and video functions. FFmpeg is the most popular open source framework used for multimedia Android development; it's a good starting point for researching Intel architecture performance software tuning. For more information about the FFmpeg project, visit <http://www.ffmpeg.org/>.

We'll start by making a new, full-format x86 player. The open source `tewilove_faplayer` is recommended for this project. It's based on the VLC player, but `tewilove_faplayer` includes all needed components, whereas the VLC player must first do a bootstrap to download all components. The VLC player's compiling script is also more complicated than `tewilove_faplayer`. The project URL is <https://github.com/shaobin0604/faplayer>, where project notes can be read and the ZIP file can be downloaded. The `tewilove_faplayer` can be easily used as is on ARM platforms; for x86, there are some necessary modifications:

1. Modify `vlc_fixups.h`. Delete `__cplusplus`, as it will otherwise lead to a compiling problem.
2. Modify `\jni\vlc\config.h`. Add the macro as defined here:

```
#define CAN_COMPILE_MMX 1
#define CAN_COMPILE_MMXEXT 1
#define CAN_COMPILE_SSE 1
#define CAN_COMPILE_SSE2 1
#define asm __asm__
#define MODULE_NAME_IS_i422_yuy2_sse2
#define MODULE_NAME_IS_i420_yuy2_sse2
#define MODULE_NAME_IS_i420_rgb_sse2
```

3. Modify `libvlcjni.h`. Delete `yuv2rgb`; it is ARM NEON code, not x86 code.
4. Modify `Application.mk` as follows:

```
APP_ABI := x86
BUILD_WITH_NEON := 0
OPT_CPPFLAGS += -frtti -fexceptions
```

5. Delete `Android.mk` in `ext\ffmpeg`; you must replace it with the x86 FFmpeg version.

Compile x86 FFmpeg: Cross-Compile

Broadly speaking, open source programs often support cross-compiling. FFmpeg is no exception. Here is a script file that you can use to build FFmpeg on Linux and Android:

```
#!/bin/bash
NDK=$ANDROID_NDK_ROOT #your ndk root path
PLATFORM=$NDK/platforms/android-14/arch-x86
#PREBUILT=$NDK/toolchains/arm-linux-androideabi-4.4.3/prebuilt/linux-x86
PREBUILT=$NDK/toolchains/x86-4.4.3/prebuilt/linux-x86
function build_one
{
./configure --target-os=linux \
--prefix=$PREFIX \
```

```

--enable-cross-compile \
--extra-libs="-lgcc" \
--arch=x86 \
--cc=$PREBUILT/bin/i686-android-linux-gcc \
--cross-prefix=$PREBUILT/bin/i686-android-linux- \
--nm=$PREBUILT/bin/i686-android-linux-nm \
--sysroot=$PLATFORM \
--extra-cflags="-O3 -fpic -DANDROID -DHAVE_SYS_UIO_H=1 -Dipv6mr_
interface=ipv6mr_ifindex -fasn -Wno-psabi -fno-short-enums -fno-strict-
aliasing -finline-limit=300 $OPTIMIZE_CFLAGS " \
--disable-shared --enable-static \
--extra-ldflags="-Wl,-rpath-link=$PLATFORM/usr/lib -L$PLATFORM/usr/lib
-nostdlib -lc -lm" \
--disable-ffplay --disable-avfilter --disable-avdevice --disable-ffprobe \
--disable-yasm \
$ADDITIONAL_CONFIGURE_FLAG

make clean
make -j4 install
}

#x86
CPU=x86
OPTIMIZE_CFLAGS="-march=atom -ffast-math -msse3 -mfpmath=sse"
PREFIX=./android/$CPU
ADDITIONAL_CONFIGURE_FLAG=
build_one

```

After running this script, you can use `libavcode.a`, `libavformat.a`, `libavutil.a`, and `libswscale.a`; link these libraries to your project as prelink static libraries.

Compile x86 FFmpeg: Android.mk

It's best to cross-compile FFmpeg; it's simple and quick. But if FFmpeg is needed to compile the `Android.mk`, this can still be done. *Havlenapetr* FFmpeg can be used to build this script.

Havlenapetr is an early Android FFmpeg project and therefore has relatively simple audio and video synchronization features. It is suitable for beginners to learn how to transplant FFmpeg on Android. Its project URL is <https://github.com/havlenapetr>.

When you have all of the tools downloaded and the preparation completed, it's time to make *faplayer x86*, which is a full format x86 player. You must first play a 1080p MP4 on the device—this will use the Android default player. For software tuning, modify the `selectMediaPlayer` function in `PlayerActivity.java` and set `useDefault` to `false`. Play the 1080p MP4 again on Android 2.3. As expected, the performance is suboptimal. Luckily, with software tuning, you can improve the performance. (If you cannot display an image on Android 4.0, see the section titled **How to Display an Image Using the Android 4.0 NDK** later in this chapter.)

How to Determine CPU Usage and Find Hotspots

Intel's Graphics Performance Analyzers (GPA) and VTune amplifier are strongly recommended as tuning tools, but if GPU usage is not the target, there are other tuning tool options, such as OProfile (which will require building a system image).

Show CPU Usage Dynamically Onscreen

CPU usage can be inquired by `/proc/stat`, using the command `cat /proc/stat` on Linux (and thus Android). The command will produce some strings like these:

```
cpu 4884 440 2841 75755 1681 320 121 0 0 0
cpu0 2211 212 1639 38296 462 223 90 0 0 0
cpu1 2673 228 1202 37459 1219 97 31 0 0 0
```

Here are some CPU usage functions written in Java. You can use them to show CPU usage onscreen:

```
/sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_max_freq
/sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_min_freq
/sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq
/proc/stat
    public static long getCurCpuFreq() {
        String result = "N/A";
        try {
            FileReader fr = new FileReader(
                /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_
freq");
            BufferedReader br = new BufferedReader(fr);
            String text = br.readLine();
            br.close();
            fr.close();
            result = text.trim();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return Long.parseLong(result)/1000;
    }
```

```

public static long getCurUsage() {
    try
    {
        BufferedReader reader = new BufferedReader( new
InputStreamReader( new FileInputStream( "/proc/stat" ) ), 1000);
        String load = reader.readLine();
        reader.close();
        String[] toks = load.split(" ");
        long currTotal = Long.parseLong(toks[2]) +
Long.parseLong(toks[3]) + Long.parseLong(toks[4])+Long.parseLong(toks[6])
+Long.parseLong(toks[7])+Long.parseLong(toks[8]);
        long currIdle =
Long.parseLong(toks[5]);

        usage =(long) ((currTotal - total) * 100.0f / (currTotal -
total + currIdle - idle));
        total = currTotal;
        idle = currIdle;
    }
    catch( IOException ex )
    {
        ex.printStackTrace();
    }
    return usage;
}

```

Get Function Running Time

1. Use function `clock()` (you must include `time.h`).
2. Use register `rdtsc`.

```

static inline uint64_t read_time(void)
{
    uint32_t a, d;
    __asm__ volatile("rdtsc" : "=a" (a), "=d" (d));
    return ((uint64_t)d << 32) + a;
}

```

These two functions return the runtime of a function.

Use Yasm to Get the Best-Performing x86 Library

Yasm is an x86 ASM assembler. In the case of FFmpeg porting, it is generally recommended that you add the `-disable-yasm` option, when you're compiling the ARM version. But if you're compiling an x86 version, `disable-yasm` will discard a large amount of optimization code, which will significantly reduce performance.

Yasm is a complete rewrite of the NASM assembler under the “new” BSD license (some portions are under other licenses; see the Yasm site at <http://yasm.tortall.net/> for details). Yasm currently supports the x86 and AMD64 instruction sets; accepts NASM and GAS assembler syntax; outputs binary, ELF32, ELF64, 32- and 64-bit Mach-O, RDOFF2, COFF, Win32, and Win64 object formats; and generates source debugging information in STABS, DWARF 2, and CodeView 8 formats.

How to Use Yasm

If Yasm is cross-compiled, it is quite simple. After the Yasm source is downloaded on Linux, and the make is installed, it is as simple as running configure, make, and make install.

```
./configure --enable-shared --prefix=/usr/local
make
make install
```

Then you run `delete -disable-yasm` and run the build script again. But if you’re using `Android.mk` directly, note that the Google NDK build script does not support `.asm`. The following link is an improvement of the Google NDK build script. You can visit <http://software.intel.com/en-us/articles/using-yasm-compiler-on-android-ndkbuild> for more details.

The Result of Using Yasm

For 1080p mp4 software decoding, Figure 12-1 compares three configurations.

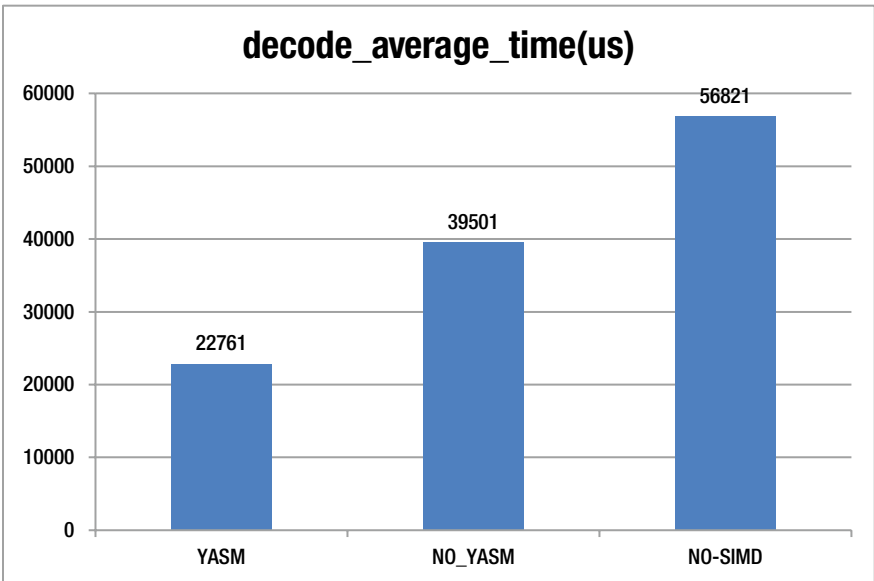


Figure 12-1. YASM Comparisons, in Nanoseconds

```
YASM: -enable yasm -enable asm
NO-YASM: -disable yasm -enable asm
NO-SIMD: -disable yasm -disable asm
```

Enabling Yasm can create significant performance improvements. In the previous example, the average time drops by 57.6% just by enabling Yasm. (The average time drops by 150% with both Yasm and ASM enabled; see the next section of this chapter for more information about SIMD and ASM.) Downloading and installing Yasm can be a valuable step for many kinds of open source optimization projects, including those on x264, Vp8, and manifestly x86.

Use SSE (Intel’s Streaming SIMD Extensions) to Optimize Color Space Transformation

The *color space* of an image represents the color in a machine-readable format. Just as different people might describe Vincent van Gogh’s *Irises* as “purple” or “indigo,” a plasma television represents the color in RGB format¹ whereas the print file for a poster of the painting uses the CMYK code.² To translate the images between these different color space formats, a color space transformation must occur. Video is generally YUV format; LCD screens are RGB format; and camera output is generally nv21 format. FFmpeg provides the `swscale` function to perform this transformation. For large image files, color space transformation will consume much more CPU power, as you can see in Table 12-1. Using Intel’s Streaming SIMD Extensions (SSE) instruction set can produce a 6-16 time performance improvement.

Table 12-1. SSE Optimizations

3040*1824 NV21-RGB888	SWS_BILINEAR	SWS_FAST_BILINEAR
Not Using Yasm	425 ms	158 ms
Using Yasm	179 ms	155 ms
Using SSE NV21-RGB888	27 ms	27 ms

SSE (SIMD technology) is the most important optimization technology on x86 Android (ARM has NEON—which is also a SIMD technology), especially for multimedia apps. This is due to the performance optimization that it provides across the board.

¹The red, green, blue (RGB) code is an additive color model based on the three primary colors of light. The color of the irises is represented as 69, 102, 137.

²The subtractive color model known as CMYK represents the four ink colors used in most commercial printing systems: cyan, magenta, yellow, and key or black. The CMYK values for the irises are 50, 26, 0, 46.

What Is SIMD?

Single instruction, multiple data (SIMD) devices have multiple processing elements that perform the same operation on multiple data points simultaneously. Most modern CPU designs include SIMD instructions in order to improve multimedia performance. Intel's Medfield CPUs support MMX, MMX2, SSE, SSE2, SSE3, and SSSE3, but SSE4 and AVX are not supported. SIMD support can also be checked dynamically at runtime (see `cputime` on FFmpeg 1.0—function `ff_get_cpu_flags_x86` on `cpu.c`).

There are three ways to implement SIMD code.

- C/C++ language-level intrinsic functions, defined with `emmintrin.h`. Until now, few open source libraries have used it except for WEBP. If SIMD code is implemented in this way, it's easy to adapt to all hardware platforms. (It becomes interchangeable with NEON code for the ARM platform, for example.)
- Inline assembler (the most widely used). It doesn't require separate assembly and link steps and is more convenient than a separate assembler.
- Separate assembler. It has many styles (NASM, TASM, MASM, and so on), and the file extension `.s`, `.asm`. (An assembler with the file extension `.asm` cannot be compiled normally by the Android NDK; you must use the patch that is provided in the earlier section, "How to Use Yasm.")

How SIMD Works

To add two 8-bit integer arrays, the generic C code looks like this:

```
int s[16];
for(int i=0;i<16;i++){
    s[i]=data1[i]+data2[i]; //ensure s[i] is 0~255
}
```

But if you use SSE, you just need:

```
movups data1,xmm1
movups data2,xmm2
paddusb xmm1,xmm2
movntq xmm2,S
```

With the one instruction `paddusb`, you can perform 16 add operations at the same time. This sounds fantastic, but it actually has limitations. All the data must be well organized, and the algorithm can be vectorized. However, this indeed improves performance, especially for multimedia apps.

SIMD has five types of instructions:

- Data movement, such as `movd`, `movq`, and `movups`
- Boolean logic: `psllw` and `psrlw`
- Math: `paddb` and `pmulhw`
- Comparisons: `pcmpeqb`
- Data packing: `packssdw`, `punpcklbw`, and `pshufb`

Data packing (see Figures 12-2 and 12-3) is the most difficult part for SIMD. In the following two figures you can see two different operations, their organization and structure, and how they perform the data packing procedure.

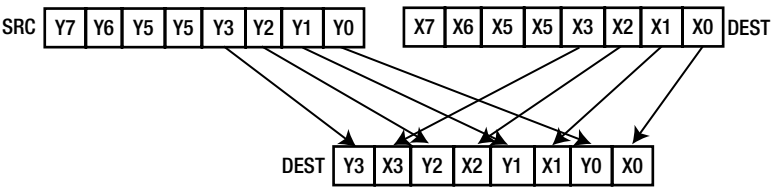


Figure 12-2. Data Packing with `Punpcklbw`

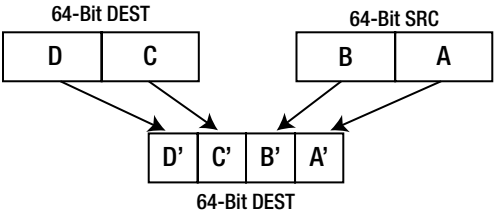


Figure 12-3. Data Packing with `Packssdw`

Packed shuffle bytes (`pshufb`) takes registers of bytes $R = [R_0 R_1 R_2 \dots R_{15}]$ and $M = [M_0 M_1 M_2 \dots M_{15}]$ and replaces R with $[R_{M_0} R_{M_1} R_{M_2} \dots R_{M_{15}}]$; except that it replaces the i th entry with 0 if the top bit of M_i is set. This is shown in the following code:

R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15
M0	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	M14	M15

```
R0 := (mask0 & 0x80) ? 0 : SELECT(a, mask0 & 0x07)
R1 := (mask1 & 0x80) ? 0 : SELECT(a, mask1 & 0x07)
...
R15 := (mask15 & 0x80) ? 0 : SELECT(a, mask15 & 0x0f)
```

The `pshufb` instruction can put any 8-bit data to any place according to a 128-bit mask.

Implement NV21-RGB SSE Code

FFmpeg yuv2rgb is MMX2 code, so you must first modify it to SSE code, because MMX is 8-bit aligned and SSE is 16-bit aligned. You must enlarge the data to 16 bits:

1. Modify `swscale_internal.h` and `yuv2rgb_mmx.c`:

```
DECLARE_ALIGNED(8, uint64_t, redDither);
==>
DECLARE_ALIGNED(16, uint64_t, redDither);
DECLARE_ALIGNED(8, uint64_t, redDither1);

DECLARE_ASM_CONST(16, uint64_t, mmx_redmask) = 0xf8f8f8f8f8f8f8ULL;
==>
DECLARE_ASM_CONST(8, uint64_t, mmx_redmask1) = 0xf8f8f8f8f8f8f8ULL;
DECLARE_ASM_CONST(16, uint64_t, mmx_redmask) = 0xf8f8f8f8f8f8f8ULL;
```

Now `redDither` and `mmx_redmask` can be used as 8-bit data or 16-bit data.

2. Change the `mov` and `mm` instructions:

```
#if HAVE_SSE2
#define MM1 "%xmm"
#define MM "%xmm"
#define MOVD "movq"
#define MOVQ "movups"
#define MOVNTQ "movntps"
#define SFENCE "sfence"
#define SIMD8 "16"
#else
#if HAVE_MMX2
#define MM1 "%mm"
#define MM "%mm"
#define MOVD "movd"
#define MOVQ "movq"
#define MOVNTQ "movntq"
#define SFENCE "sfence"
#define SIMD8 "8"
#endif
#endif
```

MMX uses a `mm` register while SSE uses an `xmm` register. Because SSE has 128-bit data length (16 bytes), the data offset is 16 when using SSE (SIMD 8 is 16).

3. `RGB_PACK24` must be rewritten due to fact that the data length of MMX and SSE are different.

```
DECLARE_ASM_CONST(16, uint8_t, rmask1[16]) = {0x00,0x80,0x80,0x01,
0x80,0x80,0x02,0x80,0x80,0x03,0x80,0x80,0x04,0x80,0x80,0x05};
...
```

```
MOVQ"      "MM"red",      "MM"5 \n"\n"
"pshufb    "MANGLE(rmask1)", "MM"5 \n"\n"
MOVNTQ"    "MM"5,      (%1) \n"\n"
```

Here, `pshufb` is used. The key idea is to use `pshufb` to put each R, G, B value into the right place and use it to get RGB888 data. The following code shows what each RGB segment is composed of. For example RGB0 is R0, G0, and B0.

R0	R1	R2	R3	R4	R5
G0	G1	G2	G3	G4	
B0	B1	B2	B3	B4	
RGB0	RGB1	RGB2	RGB3	RGB4	
R6	R7	R8	R9	R10	
G5	G6	G7	G8	G9	G10
B5	B6	B7	B8	B9	
RGB5	RGB6	RGB7	RGB8	RGB9	RGB10
R11	R12	R13	R14	R15	
G11	G12	G13	G14	G15	
B10	B11	B12	B13	B14	B15
RGB11	RGB12	RGB13	RGB14	RGB15	

4. Add `ff_nv2rgb_init_mmx`, and add `ff_nv2rgb_get_func_ptr` in the function `ff_get_unscaled_swscale`:

```
/* yuv2bgr */
if ((srcFormat==PIX_FMT_YUV420P || srcFormat==PIX_FMT_YUV422P
|| srcFormat==PIX_FMT_YUVA420P) && isAnyRGB(dstFormat)
&& !(flags & SWS_ACCURATE_RND) && !(dstH&1)) {
    c->swScale= ff_yuv2rgb_get_func_ptr(c);
}
/* nv2bgr */
if (srcFormat==PIX_FMT_NV21 && isAnyRGB(dstFormat)&& !(flags
& SWS_ACCURATE_RND) && !(dstH&1)) {
    c->swScale= ff_nv2rgb_get_func_ptr(c);
}
```

How to Display an Image Using the Android 4.0 NDK

Surface_lock and Surface_unlockAndPost can be used on the NDK layer to display an image on a device. These two functions move from libsurfaceflinger_client.so to libgui.so, and to libui.so on Android 4.0.

Using nm (Windows is nm.exe), you can perform analysis on libui.so. For example, on Windows, you can use nm.exe libui.so >>1.txt, and open 1.txt to find Surface_lock. You can find the string "_ZN7android7Surface4lockEPNS0_11SurfaceInfoEPNS_6RegionE", use it with dlsym, and then get the function handle from libui.so.

```

    clz = (*env)->GetObjectClass(env, s);
    f_Surface_mSurface = (*env)->GetFieldID(env, clz, "mSurface", "I");
    if (f_Surface_mSurface == 0)
    {
        jthrowable e = (*env)->ExceptionOccurred(env);
        if (e)
        {
            (*env)->DeleteLocalRef(env, e);
            (*env)->ExceptionClear(env);
        }
        f_Surface_mSurface = (*env)->GetFieldID(env, clz, "mNativeSurface",
"I");
    }
    (*env)->DeleteLocalRef(env, clz);
    surface = (*env)->GetIntField(env, s, f_Surface_mSurface);

```

With this, the code above the jni function can be added; it receives the Java layer surface object and gets the surface handle from the object. The f_Surface_mSurface and Surface_lock and Surface_unlockAndPost functions can be used to show an image on the NDK layer.

The Common Cross-Compile Script

Open source configuration files can generally be used to cross-compile for Android. The following is a common script for the x86 platform. It uses SSE to optimize JPEG encoding and decoding.

```

#!/bin/bash

HOSTCONF=x86
BUILDCONF=i686-pc-linux-gnu
NDK=$ANDROID_NDK_ROOT

```

```

TOOLCHAIN=$NDK/toolchains/x86-4.4.3/prebuilt/linux-x86
PLATFORM=$NDK/platforms/android-14/arch-x86
PREFIX=/home/lym/libjpeg-turbo-1.2.1/android/x86
ELF=$TOOLCHAIN/i686-android-linux/lib/ldscripts/elf_i386.x

export ARCH=x86
export SYSROOT=$PLATFORM
export PATH=$PATH:$TOOLCHAIN/bin:$SYSROOT
export CROSS_COMPILE=i686-android-linux
export CC=${CROSS_COMPILE}-gcc
export CXX=${CROSS_COMPILE}-g++
export AR=${CROSS_COMPILE}-ar
export AS=${CROSS_COMPILE}-as
export LD=${CROSS_COMPILE}-ld
export RANLIB=${CROSS_COMPILE}-ranlib
export NM=${CROSS_COMPILE}-nm
export STRIP=${CROSS_COMPILE}-strip
export CFLAGS="-I$PLATFORM/usr/include -O3 -nostdlib -fpic -DANDROID -fasm
-Wno-psabi -fno-short-enums -fno-strict-aliasing -finline-limit=300 -fomit-
frame-pointer -march=i686 -msse3 -mfpmath=sse"
export CXXFLAGS=$CFLAGS
export LDFLAGS="-Wl,-T,$ELF -Wl,-rpath-link=$PLATFORM/usr/lib -L$PLATFORM/
usr/lib -nostdlib -lc -lm"

./configure --enable-shared --host=$HOSTCONF --build=$BUILDCONF --with-
sysroot=$SYSROOT --prefix=$PREFIX
make clean
make -j4 install

```

Benefitting from the long history of the x86 platform, nearly all open source projects have done optimizations on the x86 platform, especially in the multimedia category.

A large number of arithmetic functions have been written to SIMD code

(FFmpeg), vp8, x264, and OpenCV). Generally, you need to choose the right source and use the right compiling script. When necessary, debugging the NDK assembly code is possible on Linux.

Testing and Profiling with Hardware Acceleration

With the Android 4.0 NDK, video and audio decoding are based on OpenMAX AL 1.0.1, which permits both software and hardware decoding. OpenMAX (Open Media Acceleration), developed by the Khronos Group, is a royalty-free, cross-platform API. Its set of C-language programming interfaces provides object abstractions for audio, video, and still images. It allows these resources to be easily ported across a wide range of platforms. It's intended for devices that process or consume large amounts of multimedia data, particularly embedded and mobile devices such as smartphones and tablets. OpenMAX has three layers of interfaces, as shown in Figure 12-4—the Application layer (AL), the Integration layer (IL), and the Development layer (DL).

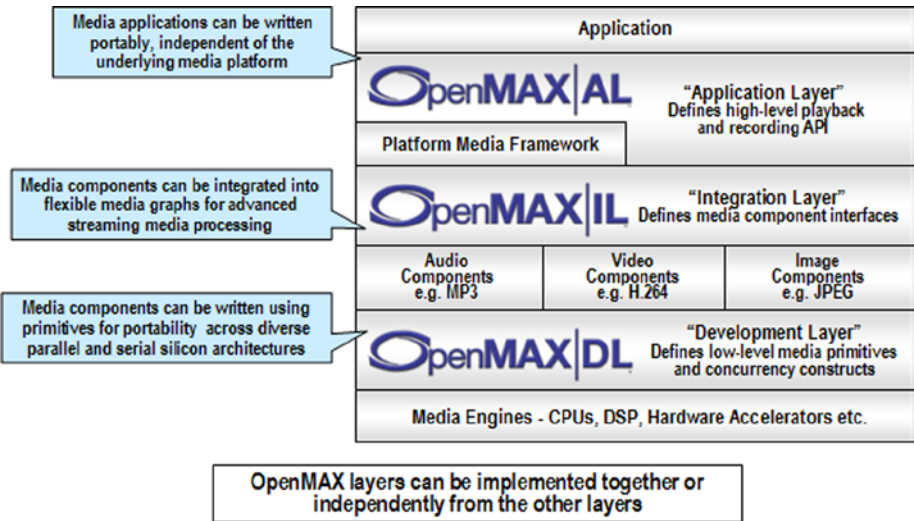


Figure 12-4. The OpenMAX Layers

Using the Integration Layer (IL) for Hardware Encoding

Before Android 4.1, Google did not expose any hardware codec interfaces to developers, so a large number of multimedia apps had to use FFmpeg, x264, and VP8 as the video codec. Especially when video encoding, software encoding will use most of the CPU resources (640×480 H.264 encoding will occupy nearly 90% of CPU resources for an ARM v9 1.2-GHz dual-core). This is an enormous drain on the Android device's performance. So up until version 4.1, no wonderful video recording apps had been developed for Android. Developer and user interest was certainly present, but such applications just weren't feasible. Now developers can use the OpenMAX Integration Layer (IL) to get a hardware encoder interface. (To clarify, both ARM and Intel architecture can use this method for hardware encoding, but ARM architectures will meet compatibility issues.) The OpenMAX IL is implemented by various vendors and Google does not guarantee its compatibility, so it's not guaranteed to work well on all Android-enabled hardware systems. But for Android on Intel architecture, specifically, the compatibility issues have been resolved.

How to Get the OMX-IL Interface on Android for Intel Architecture

`Libwrs_omxil_core_pvwrapped.so` is the OMX-IL interface layer on the Medfield Intel architecture platform. Developers can load this as follows to access the OMX-IL interface.

```
pf_init = dlsym( dll_handle, "OMX_Init" );
pf_deinit = dlsym( dll_handle, "OMX_Deinit" );
pf_get_handle = dlsym( dll_handle, "OMX_GetHandle" );
pf_free_handle = dlsym( dll_handle, "OMX_FreeHandle" );
pf_component_enum = dlsym( dll_handle, "OMX_ComponentNameEnum" );
pf_get_roles_of_component = dlsym( dll_handle, "OMX_GetRolesOfComponent" );
```

After getting these handles, you can call `pf_component_enum` and `pf_get_roles_of_component` to get the right hardware encoding interface. All of the video codec interfaces are listed here:

```
component OMX.Intel.VideoDecoder.AVC
- role: video_decoder.avc
component OMX.Intel.VideoDecoder.H263
- role: video_decoder.h263
component OMX.Intel.VideoDecoder.WMV
- role: video_decoder.wmv
component OMX.Intel.VideoDecoder.MPEG4
- role: video_decoder.mpeg4
component OMX.Intel.VideoDecoder.PAVC
- role: video_decoder.pavc
component OMX.Intel.VideoDecoder.AVC.secure
- role: video_decoder.avc
component OMX.Intel.VideoEncoder.AVC
- role: video_encoder.avc
component OMX.Intel.VideoEncoder.H263
- role: video_encoder.h263
component OMX.Intel.VideoEncoder.MPEG4
- role: video_encoder.mpeg4
```

You can choose the right component according to your needs. For example, if you want to do MP4 encoding, you can choose `OMX.Intel.VideoEncoder.MPEG4`, and call `pf_get_handle` to get the hardware MP4 encoding handle.

How Does the OMX-IL Work?

In order to create or configure and connect the OpenMAX components, the application is written as an Integration Layer (IL) client. This IL client is used to invoke OpenMAX APIs for different components, as represented in Figure 12-5. In this application,

components allocate the video buffers in response to OMX APIs on the IL client. The IL client is responsible for taking the buffers from one component and passing them to other components. The functions `OMX_GetParameter` and `OMX_SetParameter` are used as a parameter/configuration set and get. `OMX_SendCommand` is used to send commands to a component, including an enable/disable port command and a state change command. `OMX_EmptyThisBuffer` and `OMX_FillThisBuffer` pass the buffers to components. `OmxEventHandler`, `OmxEmptyBufferDone`, and `OmxFillBufferDone` (`OMX_CALLBACKTYPE`) must be registered when calling `pf_get_handle`.

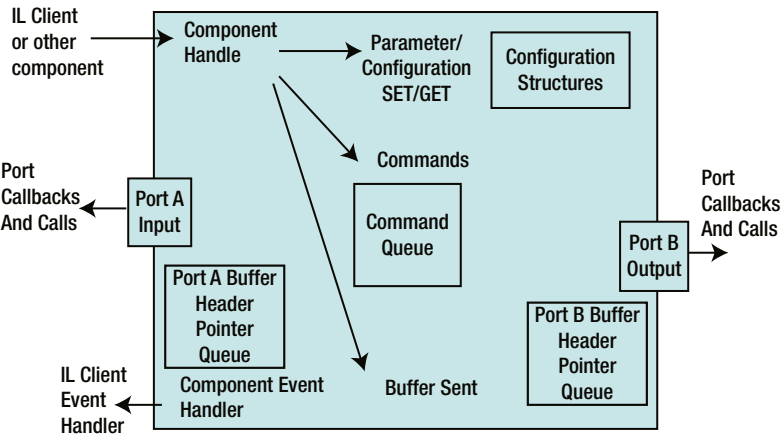


Figure 12-5. The OpenMAX components and the Integration Layer Client

After allocating the OMX buffer and calling `OMX_SendCommand` to set the `OMX_StateExecuting` state, you can use `FillThisBuffer` and `EmptyThisBuffer` and their callback functions to do hardware encoding. Figure 12-6 shows the call sequence.

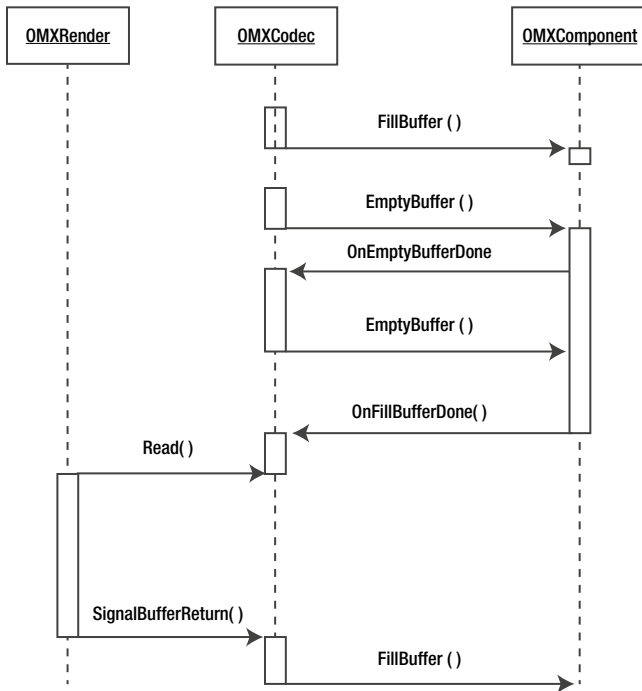


Figure 12-6. *The OMX-IL Rendering Pipeline*

Call `FillThisBuffer` to fill one raw picture into the OMX local buffer and call `EmptyThisBuffer` to make the OMX component do the hardware encoding; when you're finished encoding or when the local output buffer is full, the OMX component will call `OnEmptyThisBufferDone` to tell the client to do `EmptyThisBuffer` again. So one `FillThisBuffer` may make several instances of `OnEmptyThisBufferDone`. If the OMX component finds that the input buffer is empty, it will call `OnFillThisBufferDone` to tell the client to fill more buffer space.

Demo: Special Effects Video Recorder

In this section, you use a hardware video encoder to implement a special effects video recorder. The idea is to get data from the camera and add a mark to the camera preview image. When recoding, the mark should be recorded into the video file. It sounds like a simple design, but before Android 4.1, the only choice was a software encoder (FFmpeg or x264), which would waste lots of CPU resources. Starting with Android 4.1, a new class `MediaCodec` has been introduced; it's the same as OMX-IL, but implemented by Google, and Google guarantees its compatibility.

Table 12-2 represents a demonstration of the CPU usage for three video recorders. The recorder file format is .MP4, which uses an MPEG-4 encoder. Generally, VP8 video encoding requires the least CPU usage, followed by MPEG-4 encoding, while H.264 encoding will cost the most CPU resources).

Table 12-2. *Hardware vs. Software Encoders*

Video Recorder	Frequency	CPU Usage	Total Resource
1024x576 preview	600 MHz	35%	35% x 600/1600 = 13.125%
Hardware Encoder Without Sound	600 MHz	45%	45% x 600/1600 = 16.875%
Software Encoder Without Sound	1600 MHz	50%	50% x 1600/1600 = 50%

If you’re using a hardware encoder, total CPU resources needed are just 3.7% (hardware encoding – preview), whereas a software encoder will need 46.9%. The resolution used in this example is 1024×576. If you want 1080P video recording, the software solution is impossible!

Packaging a Hardware Video Encoder Library

The following code shows a dynamic library named `libomx.so`. It provides three simple functions—`EncInit`, `EncVideo`, and `EncRelease`. The usage is also simple—you call `EncInit` to initialize the hardware encoder, call `EncVideo` to do the hardware encoding, and call `EncRelease` to release the hardware encoder. The two main structures are `stEncConfig` and `stEncPic`:

```
- stEncConfig (use in EncInit)
-     stcfg.id = ENC_MPEG4;           //choose the encoder
-     stcfg.type = ENC_DEFAULT;       //for feature use, now must this value
-     stcfg.w=1080;                   //encoding size
-     stcfg.h=1920;
-     stcfg.framerate = 15;           //encoding framerate
-     stcfg.controlrate = enum OMX_VIDEO_CONTROLRATETYPE;
-     stcfg.bitrate = xxxx;           //your bitrate

- stEncPic(use in EncVideo)
-     pic.w=1080;                     //picture size
-     pic.h=1920;
-     pic.stride = 1080;               //picture stride
-     pic.pbuf[0]=pmem;               //yuv420 image data
-     pic.pbuf[1]=pmem+1920*1080;
-     pic.pbuf[2]=pmem+1920*1080/4;
```

Implement Camera Preview

Because you cannot use the common Google Android API to develop this demo, you must implement camera preview. I recommend using `setPreviewCallbackWithBuffer` to get the camera preview data. Although `setPreviewCallbackWithBuffer` and `setPreviewCallback` can both get the preview data, the first is more efficient and will avoid Java garbage collection.

Guideline: Hold enough memory for frequent use. Doing so will avoid dummy Java garbage collection.

In the preview callback, you should pass image data to the NDK layer; do not add unnecessary code. I have added `camera.getParameters().getPreviewSize().width` as an example to show that it costs a significant amount of CPU usage.

Guideline: Call any object function as seldom as possible. Assign its value to a variable and use the variable instead.

In the NDK layer, transform the preview data from NV21 to RGB565 (or RGB88, according to your screen configuration) and then display the data to the screen (use `Surface_lock` and `Surface_unlockAndPost`).

Profiling Java Code with Traceview

You can use Traceview to analyze the performance of Java code. Traceview has been integrated into Eclipse through a new plug-in. The plug-in integrates with the DDMS plug-in so that using the start/stop profiling button will open traces in Eclipse directly instead of launching the standalone tool. Additionally, if you Ctrl-click (Command-click on a Mac) on a method it will open the source file in the editor (you must add `android:debuggable="true"`).

Choose the right package name, which is `com.Filters`, and Press Start Method Profiling, as shown in Figure 12-7. Wait for a while and then stop the profiling: you can see the trace result in Figure 12-8.

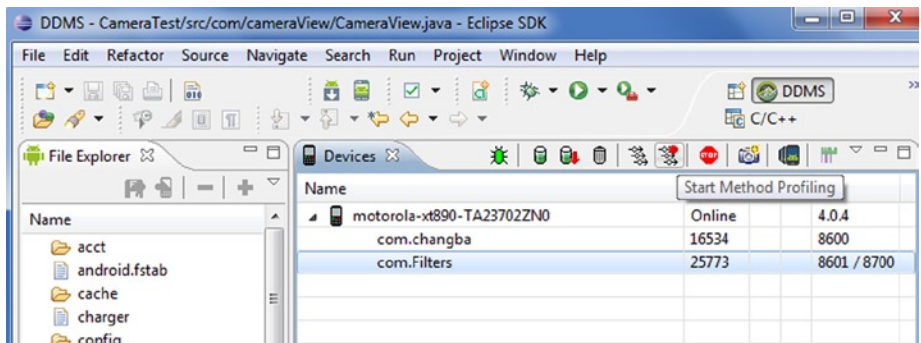


Figure 12-7. Using Filters in Eclipse

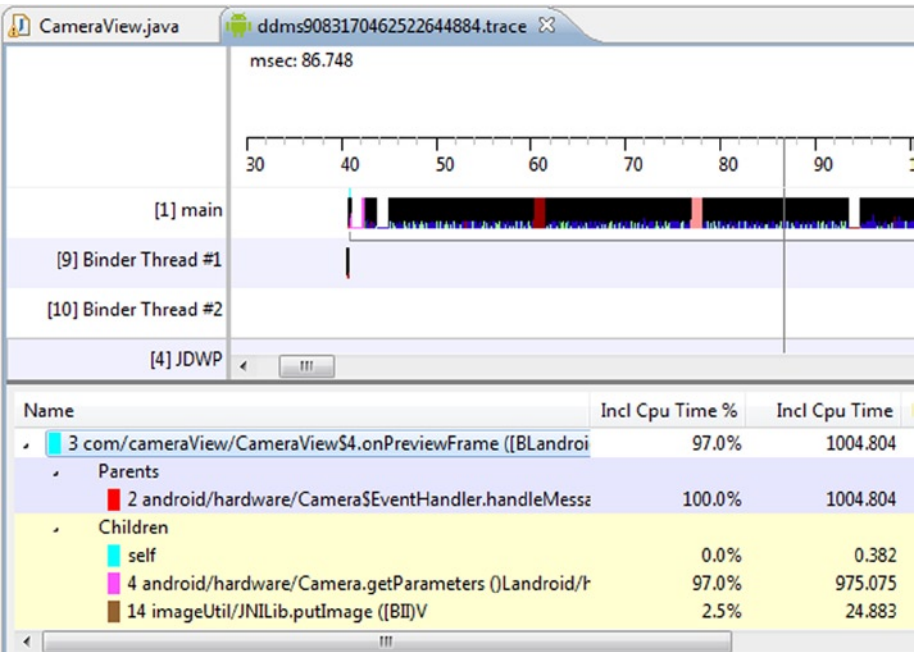


Figure 12-8. Profiling the Camera Application

From the result, it is easy to see that `Camera.getParameters` will cost most of the CPU resources (97%). That is `camera.getParameters().getPreviewSize().width` in `PreviewCallback`. Traceview can only profile Java code. If you want to profile NDK code, you can use the Intel tool VTune, which you can get from <http://software.intel.com/en-us/intel-vtune-amplifier-xe>.

Start a Recording Thread

To reference output from `-example.c` in FFmpeg, you can use this as a base version. The remaining work is determining how to get the video and sound data in a recording thread. The general idea is that once you get one frame of video (or audio), you can lock the buffer and invoke a recording thread to start working. However, this is very inefficient. The parallelism of the camera preview thread, the audio thread, and the hardware recording thread has been destroyed, and most of time, the CPU is just waiting. This is where the `CircleBuffer` class comes in; the producer (camera preview thread, audio thread) will cautiously fill the buffer into `CircleBuffer` (if the buffer is full, just overlap the last buffer, so that the data will always refresh even when the recording thread is slow), and the consumer will cautiously get the buffer from it (if the buffer is null, you can choose getting the previous buffer or waiting).

Guideline: Keep the parallelism for all threads as much as possible. CircleBuffer is generally a good choice.

Adding Special Effects

Now, video raw data has been processed on the NDK layer, so you can easily add special effects such as a video mark. Simply alpha-blend your mark image (Y data, discard UV data) into video raw data. This work can be done by the CPU and also by the GPU. In fact, Google video recording also supports limited (very limited) video effects. It uses the GPU instead of the CPU to decrease CPU load. GPU rendering (OpenGL-ES) is very complicated and hard to understand. Even though the code is simple, developers should have an in-depth understanding of OpenGL. Here I just explain a basic work flow for GPU rendering.

1. Get and initialize the default EGL display.

```
EGLDisplay eglDisplay = eglGetDisplay(EGL_DEFAULT_DISPLAY);
eglInitialize(mEglDisplay, &majorVersion, &minorVersion);
```

2. Create an integer array with one element. It will hold the return value that indicates the number of EGL configurations that matched the attributes specified by the configAttributes array. Create an EGLConfig array with one element to store the first EGL configuration that matches the attributes. Invoke eglChooseConfig() and provide, as arguments, the EGLDisplay object that you initialized in Step 1, the array that specifies the configuration attributes to match, a placeholder for the first matching EGLConfig object, the size of the EGLConfig placeholder, and the num_configs array to store the number of configurations that matched. Store the single configuration from the eglConfigs array in the EGLConfig variable eglConfig.

```
EGLint configAttribs[] = {
    EGL_SURFACE_TYPE, EGL_WINDOW_BIT,
    EGL_RENDERABLE_TYPE, EGL_OPENGL_ES2_BIT,
    EGL_RED_SIZE, 8,
    EGL_GREEN_SIZE, 8,
    EGL_BLUE_SIZE, 8,
    EGL_NONE
};
eglChooseConfig(mEglDisplay, configAttribs, &config, 1, &numConfigs);
```

3. Invoke eglCreateWindowSurface() to create an EGL surface and provide, as arguments, eglDisplay and eglConfig, which are the instances of EGLDisplay and EGLConfig that you set up in Steps 1 and 2. In the following code sample, eglCreateWindowSurface() is invoked from a class that is derived from the Screen class, and this argument binds the EGLSurface object to the current screen.

```
eglCreateWindowSurface(mEglDisplay, config, mNativeWindow.get(), NULL);
```

4. Invoke `eglCreateContext()` to create an EGL context.

```
eglCreateContext(mEglDisplay, config, EGL_NO_CONTEXT, contextAttribs);
```

5. Invoke `eglMakeCurrent()` to bind the EGL context to the EGL surface and EGL display.

```
eglMakeCurrent(mEglDisplay, mEglSurface, mEglSurface, mEglContext);
```

6. Create the programs and load shader into EGL. Your special effect function needs to be implemented as a shader.

```
loadShader(GL_FRAGMENT_SHADER, fSrc[i], &fShader);
createProgram(vShader, fShader, &mProgram[i]);
```

7. Do the rendering.

```
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, mFrameWidth,
mFrameHeight, GL_RGB, GL_UNSIGNED_SHORT_5_6_5, pixels);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, 0);
```

8. If you want to read an image from EGL, you can call `glReadPixels`.

Actually, you can also use `GLSurfaceView`. A `GLSurfaceView` provides the following features:

- Manages a surface, which is a special piece of memory that can be composited into the Android view system.
- Manages an EGL display, which enables OpenGL to render into a surface.
- Accepts a user-provided `Renderer` object that does the actual rendering.
- Renders on a dedicated thread to decouple rendering performance from the UI thread.
- Supports both on-demand and continuous rendering.
- Optionally wraps, traces, and/or error-checks the renderer's OpenGL calls.

For more information about `GLSurfaceView`, check <http://developer.android.com/reference/android/opengl/GLSurfaceView.html>. You can find a sample of it at <http://www.learnopengles.com/how-to-use-opengl-es-2-in-an-android-live-wallpaper>.

Use OpenMAX AL on Android 4.0

The OpenMAX AL API provides application-level multimedia solutions with portability across an array of platforms by providing a common abstraction for a system's media playback and recording functionality. The API organizes this abstraction around a set of high-level objects. An application acquires all objects from one “engine” object, which encapsulates an OpenMAX AL session and serves as an umbrella for all other objects.

The Advantage of Using Native Multimedia API (OpenMAX AL)

Starting with Android 4.0, Google included the Android native multimedia APIs based on the Khronos group OpenMAX AL 1.0.1 standard, as of Android API level 14 (Android platform version 4.0) and higher. It provides a direct, efficient path for low-level streaming multimedia. The new path is ideal for applications that need to maintain complete control over media data before passing it to the platform for presentation.

For example, media applications can now retrieve data from any source, apply proprietary encryption/decryption, and then send the data to the platform for display. Applications can also now send processed data to the platform as a multiplexed stream of audio/video content in MPEG-2 transport stream format. The platform demuxes, decodes, and renders the content. The audio track is rendered to the active audio device, while the video track is rendered to either a Surface or a SurfaceTexture. When rendering to a SurfaceTexture stream format, the application can apply subsequent graphics effects to each frame using OpenGL.

■ **Note** Although it's based on OpenMAX AL, the Android native multimedia API is not a conforming implementation of either OpenMAX AL 1.0.1 profile (media player or media player/recorder). This is because Android does not implement all of the features required by either of the profiles. Any known cases where Android behaves differently than the specification are described in the section “Android Extensions” that follows. The Android OpenMAX AL implementation has limited features and is intended primarily for certain performance-sensitive native streaming multimedia applications, such as video players. Table 12-3 indicates objects and interfaces supported by Android's OpenMAX AL implementation. A shaded cell means the feature is supported.

Table 12-3. *Objects and Interfaces Supported by Android's OpenMAX AL Implementation*

Objects and Interfaces	Engine	Media Player	Output Mix
Dynamic interface management			
Engine			
Object			
Play			
Prefetch status			
Stream information			
Video decoder capabilities			
Volume			
Native display data locator		Video sink	
Output mix data locator		Audio sink	
MIME data format		source	

Demo: Streaming Media Player

Google has provided a sample for a streaming media player. You can check the sample in `samples\native-media\jni \native-media-jni.c` in your `android-sdk` folder. The function `Java_com_example_nativemedia_NativeMedia_createStreamingMediaPlayer` will create a native streaming media player.

The usage is very simple, but you should note two points:

- The video source must be a `NativeWindow`. You can call `ANativeWindow_fromSurface` to get the native window from `Surface`, which passes from the Java layer. And the `Surface` must be a `GLSurfaceView` to ensure hardware rendering.
- Register `XA_ANDROIDBUFFERQUEUEEVENT_PROCESSED` callback for filling the streaming buffer. The demo is reading the buffer from a file, but you can also read the buffer from the Internet, so that you can implement a P2P streaming media player.

Use a Powerful Media API: MediaCodec on Android 4.1

Android has a great media library that allows all sorts of powerful actions. Until recently, there was no way to encode and decode audio/video, which gives developers the ability to do almost anything. Fortunately, the Jelly Bean release introduced the `android.media.MediaCodec` API. It is designed following the same principles and architecture of OpenMAX (a well-known standard in the media industry), transitioning from a pure high-level media player to the encoder/decoder level.

Sample Code: Audio Decoder

This sample code shows how to implement a decoder. It uses two classes—`MediaCodec` and `MediaExtractor`. `MediaExtractor` facilitates extraction of demuxed, typically encoded, media data from a data source. `MediaCodec`, of course, is used as a low-level codec.

First you should use `MediaExtractor` to get the media format:

```
MediaExtractor extractor = new MediaExtractor();
extractor.setDataSource(sampleFD.getFileDescriptor(), sampleFD.
getStartOffset(), sampleFD.getLength());
MediaFormat format = extractor.getTrackFormat(0);
```

Secondly, you can create `MediaCodec` and configure it.

```
MediaCodec codec;
ByteBuffer[] codecInputBuffers;
ByteBuffer[] codecOutputBuffers;

MediaCodec codec = MediaCodec.createByCodecName(name);
codec.configure(format, null, null, 0); //no display, so surface is null
codec.start();
```

Finally, you do the decoding. Like OMX-IL, it has two ports. You should call `dequeueInputBuffer` to send the decoding buffer to `MediaCodec`, and call `dequeueOutputBuffer` to receive the outside buffer.

```
int inputBufIndex = codec.dequeueInputBuffer(TIMEOUT_US);
if (inputBufIndex >= 0) {
    ByteBuffer dstBuf = codecInputBuffers[inputBufIndex];
    int sampleSize = extractor.readSampleData(dstBuf, 0);
    long presentationTimeUs = 0;
    if (sampleSize < 0) {
        sawInputEOS = true; sampleSize = 0;
    } else {
        presentationTimeUs = extractor.getSampleTime();
    }
    codec.queueInputBuffer(inputBufIndex, 0, sampleSize,
        presentationTimeUs,
        sawInputEOS ? MediaCodec.BUFFER_FLAG_END_OF_STREAM : 0);
    if(!sawInputEOS){
        extractor.advance();
    }
}
final int res = codec.dequeueOutputBuffer(info, TIMEOUT_US);
if(res >= 0){
    int outputBufIndex = res;
    ByteBuffer buf = codecOutputBuffers[outputBufIndex];
```

```

        final byte[] chunk = new byte[info.size];
        buf.get(chunk);
        buf.clear();
        if(chunk.length > 0){
            audioTrack.write(chunk,0,chunk.length);
        }
        codec.releaseOutputBuffer(outputBufIndex, false);
        if((info.flags && MediaCodec.BUFFER_FLAG_END_OF_STREAM) != 0){
            sawOutputEOS = true;
        }
    } else if(res == MediaCodec.INFO_OUTPUT_BUFFERS_CHANGED){
        codecOutputBuffers = codec.getOutputBuffers();
    } else if(res == MediaCodec.INFO_OUTPUT_FORMAT_CHANGED){
        final MediaFormat oformat = codec.getOutputFormat();
        mAudioTrack.setPlaybackRate(oformat.getInteger(MediaFormat.KEY_SAMPLE_
RATE));
    }
}

```

Use MediaCodec in NDK

MediaCodec is a Java layer class, but you must do the decoding (or encoding) in C code, which is in the NDK layer. Therefore, it is important to call the Java class in the NDK. This can be done by using the jni function FindClass.

Sample code for FindClass is shown here:

```

jclass audio_record_class = jni_env->FindClass("android/media/AudioRecord");
int size = jni_env->CallStaticIntMethod(audio_record_class
,jni_env->GetStaticMethodID(audio_record_class,"getMinBufferSize", "(III)I")
,preferred_rate
,2/*CHANNEL_CONFIGURATION_MONO*/
,2/* ENCODING_PCM_16BIT */);

```

Configuring MediaCodec with a surface will provide the best performance, as it can use hardware rendering. You can reference native media (for example, OpenMAX AL) to add the surface (or GLSurfaceView). If you're using a normal surface, you can use a textview for subtitles, and an imageview for the play bar. If you're using GLSurfaceView, you can extend this class and implement your own renderer.

Overview

There are currently only two hardware acceleration technologies that can be used on the Android app layer: OpenGL and OpenMAX. OpenGL includes GLSurfaceView and OpenGL-ES 1.0 and 2.0 in the NDK, and it is generally used as a renderer or for multimedia effects processing. OpenMAX includes OpenMAX AL in NDK, MediaCodec, and OMX-IL (which is not Google code, but must be implemented by the author). Each

technology has a usage scenario and applicable Android version. Up until now, the popular Android versions were 2.3 and 4.0, so this chapter has only covered the versions from Android 2.3 through Android 4.1. Table 12-4 indicates which hardware accelerators can be used with which version of Android.

Table 12-4. *Hardware Accelerators Compatibility with Android Versions*

	OMX-IL	OMX-AL	MediaCodec
2.3 audio			
2.3 video decode			
2.3 video encode	Codec (Intel architecture only)		
4.0 audio		Player (NDK only)	
4.0 video decode		Player (NDK only)	
4.0 video encode	Codec (Intel architecture only)		
4.1 audio			Codec
4.1 video decode			Codec
4.1 video encode	Codec (Intel architecture only)		Codec

OpenGL is complex but the usage scenario is relatively fixed (video effect and image processing), so the table lists only the usage of OpenMAX.

The Android 4.1 MediaCodec is an important update for multimedia apps. It gives apps the ability to process images before encoding or after decoding. Using Android on Intel architecture grants this ability even before Android 4.1, and after 4.1, hardware acceleration can still give better and better effects.